INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

UniWiki: A Reliable and Scalable Peer-to-Peer System for Distributing Wiki Applications

Gérald Oster — Pascal Molli — Sergiu Dumitriu — Rubén Mondéjar

N° 6848

Février 2009

Thème COG

ISSN 0249-6399 ISRN INRIA/RR--6848--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

UniWiki: A Reliable and Scalable Peer-to-Peer System for Distributing Wiki Applications

Gérald Oster*, Pascal Molli*, Sergiu Dumitriu†, Rubén Mondéjar‡

Thème COG — Systèmes cognitifs
Équipe-Projet ECOO

Rapport de recherche n° 6848 — Février 2009 — 18 pages

Abstract: The ever growing request for digital information raises the need for content distribution architectures providing high storage capacity, data availability and good performance. While many simple solutions for scalable distribution of quasi-static content exist, there are still no approaches that can ensure both scalability and consistency for the case of highly dynamic content, such as the data managed inside wikis. In this paper, we propose a peer to peer solution for distributing and managing dynamic content, that combines two widely studied technologies: distributed hash tables (DHT) and optimistic replication. In our "universal wiki" engine architecture (UniWiki), on top of a reliable, inexpensive and consistent DHT-based storage, any number of front-ends can be added, ensuring both read and write scalability. The implementation is based on a Distributed Interception Middleware, thus separating distribution, replication, and consistency responsibilities, and also making our system usable by third party wiki engines in a transparent way. UniWiki has been proved viable and fairly efficient in large-scale scenarios.

Key-words: Collaborative editing, Optimistic replication, Peer-to-peer, Distributed Interception

* Project-Team ECOO, LORIA, INRIA Nancy - Grand Est, Nancy-Universié, {oster,molli}@loria.fr
† XWiki SAS, Nancy-Universié, sergiu@xwiki.com
‡ Universitat Rovira i Virgili, Spain, ruben.mondejar@urv.cat

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

UniWiki: Une architecture fiable et supportant le passage à l'échelle pour la distribution d'applications orientées wiki

Résumé : L'utilisation grandissante des documents numériques soulève un besoin en de nouvelles architectures proposant une immense capacité de stockage tout en garantissant une haute disponibilité des données stockées. Bien que de nombreuses propositions aient été réalisées pour répondre à cette demande pour la distribution de données quasi-immuables, il n'existe pas ou peu de proposition qui s'intéresse à ce problème pour des données hautement dynamiques telles que les données gérées par les serveurs wiki. Dans cet article, nous proposons une architecture pair-à-pair pour la distribution et la gestion de contenu hautement dynamique. Cette solution repose sur une table de hachage distribuée combinée avec un mécanisme de réplication optimiste. Afin de valider cette proposition, nous présentons une implémentation réalisée en utilisant un canevas d'interception distribué qui permet d'une part de séparer les responsabilités de distribution, de réplication et de maintien de la cohérence, et également, d'intégrer notre approche de manière transparente aux serveurs d'applications wiki existants. Cette implémentation s'est montrée viable et efficace dans des scénarios d'expérimentation à large échelle.

Mots-clés : Édition collaborative, Réplication optimiste, Réseaux pair-à-pair, Interception distribuée

Contents

1 Introduction

Peer to peer (P2P) systems, which account for a significant part of all internet traffic, rely on content replication at more than one node to ensure scalable distribution. This approach can be seen as a very large distributed storage system, which has many advantages, such as resilience to censorship, high availability, virtually unlimited storage space [2].

Currently, P2P networks mainly distribute immutable contents. We aim at making use of their characteristics for distributing dynamic, editable content. More precisely, we propose to distribute updates on this content and manage collaborative editing on top of such a peer to peer network. We are convinced that, if we can deploy a group editor framework on a P2P network, we open the way for P2P content editing: a wide range of existing collaborative editing applications, such as CVS and Wikis, can be redeployed on P2P networks, and thus benefit from the availability improvements, the performance enhancements and the censorship resilience of P2P networks.

Our architecture targets heavy-load systems, that must serve a huge number of requests. An illustrative example is Wikipedia [34], the collaborative encyclopædia that has collected, until now, over 11,500,000 articles in more than 250 languages. It currently registers at least 350 million page requests per day, and over 300,000 changes are made daily [36]. To handle this load, Wikipedia needs a costly infrastructure [4], for which hundreds of thousands of dollars are spent every year. A P2P massive collaborative editing system would allow to distribute the service and share the cost of the underlying infrastructure.

Existing aproaches to deploy a collaborative system on a distribted network include Wooki [33], DistriWiki [18], RepliWiki [26], Distributed Version Control systems [1, 11, 37], DTWiki [8] and Piki [19]. Several drawback prevent these systems from being used in our target scenario. They either require total replication of content, requiring all wiki pages are replicated at all nodes, or do not provide support for all the features of a wiki system such as page revisions, or provide only a basic conflict resolution mechanism that is not suitable for collaborative authoring.

This paper presents the design and the first experimentations of a wiki architecture that:

- is able to store huge amounts of data,
- runs on commodity hardware by making use of peer to peer networks,
- does not have any single point of failure, or even a relatively small set of points of failure,
- is able to handle concurrent updates, ensuring eventual consistency.

To achieve these objectives, our system relies on the results of two intensively studied research domains, *distributed hash tables* [27] (DHT) and *optimistic replication* [28]. At the storage system level, we use DHTs, which have been proved [14] as quasi-reliable even in test cases with a high degree of churning and network failures. However, DHTs alone are not designed for supporting consistently unstable content, with a high rate of modifications, as it is the case with the content of a wiki. Therefore, instead of the actual data, our system stores in each DHT node *operations*, more precisely the list of changes that produce the current version of a wiki document. It is safe to consider these changes as the usual static data stored in DHTs, given that an operation is stored in a node independently of other operations, and no actual modifications are performed on it. Because the updates can originate in various sources, concurrent changes of the same data might occur, and therefore different operation lists could be temporarily available at different nodes responsible for the same data. These changes need to be combined such that a plausible most recent version of the content is obtained. For this purpose, our system uses an optimistic consistency maintenance algorithm, WOOT [21], which guarantees eventual consistency, causal consistency and intention preservation [31].

To reduce the effort needed for the implementation, and to make our work available to existing wiki applications, we built our system using a distributed interception middleware (Damon [16, 17]). Thus, we were able to reuse existing implementations for all the components needed, and integrate our method transparently.

Section 2 presents approaches related to our proposition and analyzes their strong and weak points. In the same section, an overview of DHT characteristics and optimistic consistency maintenance algorithms is presented. The paper further describes, in Section 3, the architecture of the UniWiki system and its algorithms. An implementation of this system is presented in Section 4. In Section 5, a validation via experimentation on a large-scale scenario is demonstrated. The paper concludes in Section 6.

2 Related Work

2.1 Existing Approaches

One of the most relevant architectural proposals in the field of large-scale collaborative platforms is a semi-decentralized system for hosting wiki web sites like Wikipedia, using a collaborative approach. This design focuses on distributing the pages that compose the wiki across a network of nodes provided by individuals and organizations willing to collaborate in hosting the wiki. The paper [32] presents algorithms for page placement so that the capacity of the nodes is not exceeded and the load is balanced, and algorithms for routing client requests to the appropriate nodes.

In this architecture, only the storage of content wiki pages is fully distributed. Client requests (both read and write) are handled by a subset of *trusted* nodes, and meta-functionalities, such as user pages, searching, access controls and special pages, are still dealt with in a centralized manner.

While this system can resist random node departures and failures (one of the inherent advantages of replication), it does not handle more advanced failures, such as partitioning, and performs poorly with respect to concurrent updates (since write are done on limited number of trusted nodes).

Approaches that relate more to the P2P model include:

- DistriWiki [18], based on the JXTA™[12] protocol, provides a peer-to-peer wiki architecture, where each node represents a set of wiki pages stored on a user's computer. It concentrates on the communication aspect, but ignores wiki specific features, such as versioning, and does not solve the problem of merging concurrent contributions from different users.
- DTWiki [8] uses delay tolerant network (DTN) [10] as the basis for building a distributed and replicated storage mechanism, enabling offline work and access from terminals with intermittent Internet access.
- Piki [19] adapts Key Based Routing [6] and Distribute Hash Tables [27] (DHTs) to build its storage mechanism, where each node in the P2P network is responsible for storing a number of pages.
- Another approach [23] proposes the usage of DHTs as the underlying storage mechanism, backed by a safe distributed transaction manager based on Paxos, inheriting all the advantages of DHTs: quasi-reliable distributed storage, fault tolerance, censorship resilience.

These systems concentrate on the communication and replication aspects, but do not properly address concurrent updates. They either do not consider the problem, or take a transactional approach where only one user can successfully save his changes. Both approaches are clearly not adapted to collaborative authoring since some user's contributions might be lost.

Concurrent editing can be identified as the main concern of other systems:

- RepliWiki [26] aims at providing an efficient way of replicating large scale wikis like Wikipedia, in order to distribute the load and ensure that there is no single point of failure. The central part of the RepliWiki is the Summary Hash History (SHH) [13], a synchronization algorithm that aims at ensuring divergence control, efficient update propagation and tamper-evident update history.
- Wooki [33] is a wiki application built on top of a unstructured peer-to-peer network. Propagation of updates among peers is accomplished by the means of a probabilistic epidemic broadcast, and merging of concurrent updates is under control of the WOOT [21] consistency maintenance algorithm. This algorithm ensures convergence of content and intention preservation [31] – no user updates are lost in case of merging.
- Git [11], a distributed version control system (DVCS) where the central source repository is replaced by a completely decentralized peer to peer network, stands as the underlying storage mechanism for several wiki systems: git-wiki, eWiki, WiGit, Ikiwiki, ymcGitWiki. The advanced merging capabilities of Git make these wiki systems suitable for offline and distributed content authoring. Unfortunately, DVCSs ensure only causal consistency, which means convergence of replicated content is not guaranteed.

Unlike the previous approaches, the latter handle concurrent updates efficiently. However, by employing total replication, where all nodes contain copies of the whole data, they limit their scalability since the number of possible "donors" is drastically reduced by the need to be powerful enough to host the entire wiki system. They are suitable for smaller wikis, and can apply to the scenario of collaborative writing in mobile environments.

2.2 Background

Distributed Hash Tables (DHT), one of the most well discussed topics regarding P2P systems, provide a simple distributed storage interface, with *decentralization*, *scalability* and *fault tolerance* as its main properties.

Throughout the last decade, several architectures were proposed, each with its various strong points. The most notable are Chord [29], one of the favorites in the academic/research environment, Pastry [27], frequently used in practice, Tapestry [38], CAN [25] and Kademlia [15]. All DHT systems have in common a simple map storage interface, allowing to **put(key, value)** and **get(key)** back values from the storage. Beyond this common ideology, the implementations have critical differences. While some allow storing just one value for each key, others maintain a list of values, which is updated on each call of **put**. Several implementations even have an expiration mechanism, where a value is associated with a key for a limited period of time, after which it is removed from the list of values for that key.

DHTs have been successfully used in peer to peer applications, not only in file-sharing programs like BitTorrent [24], but also inside critical systems such as the storage mechanism of the heavily-accessed Amazon [7].

Although DHTs have been popular long before any of the distributed wiki systems have been proposed, few architectures employ them in wiki applications. This is a natural consequence of the fact that only static content can be reliably stored in a DHT, or at most content with infrequent updates originating from one point. Decentralized concurrent updates could create inconsistent states and cause loss of content.

A control mechanism suitable for maintaining consistency of shared data in collaborative editing is provided by the *operational transformation* approach [9, 30]. Instead of sharing the actual document, the peers maintain the list of actions that users performed to create that content. During reconciliation phases, the concurrent actions are transformed regarding each other in order to compute a document state that combines modifications performed by users. This combination is computed in such a way that it ensures convergence of copies regardless of the order in which operations are received. In comparison, traditional consistency algorithms used in version control systems – distributed or not – ensure only causal consistency: all actions are seen in the same order they have been performed, but do not guarantee convergence of copies in presence of concurrent actions.

Moreover, OT approaches ensure an additional correctness criteria called *intention preservation* [31]. Intention preservation guarantee that effect of operations are preserved while obtaining convergence. User contributions are mixed in order to compute the convergence state rather than being simply discarded.

Any kind of content can be managed using operational transformations, not only linear text, as long as the correct transformations are defined. The difficulty resides in finding these transformations. Several algorithms have been proposed based on this approach, working on environments with different specifics. Unfortunately, most of them require either a central server or the use of vector clocks in order to ensure causality between updates. These requirements limit the potential scalability of OT-based systems.

Another approach is *WOOT*, (WithOut Operational Transformation) [21], based on the same principles as operational transformation, but sacrificing the breadth of the supported content types to gain simplicity. WOOT algorithm does not require central servers nor vector clocks, and uses a simple algorithm for merging operations. However, it only applies to linear text structures and supports a reduced set of operations: *insert* and *delete*. Basically, the algorithm works[1] as follows.

WOOT sees a document as a sequence of blocks, or elements, where a block is a unit of the text, with a given granularity: either a character, word, sentence, line or paragraph. Once created, a block is enhanced with information concerning unique identification and precise placement even in a highly dynamic collaborative environment, thus becoming a *W-character*. Formally, a W-character is a five-tuple $< id, \alpha, v, id_{cp}, id_{cn} >$, where:

- id is the identifier of the W-character, a pair (ns, ng), consisting of the unique ID of the peer where the block was created, and a logical timestamp local to that peer;
- α is the alphabetical value of the block;
- v indicates whether the block is visible or not;
- id_{cp} is the identifier of the previous W-character, after which this block is inserted;
- id_{cn} is the identifier of the following W-character.

[1] For a more detailed description, please refer to [21]

When a block is deleted, the corresponding W-character is not deleted, but its visibility flag is set to *false*. This allows future blocks to be correctly inserted in the right place on a remote site, even if the preceding or following blocks have been deleted by a concurrent edit. W-characters corresponding to the same document form a partially ordered graph, a W-string, which is the model on which WOOT operates.

Every user action is transformed into a series of WOOT operations, which include references to the affected context (either predecessor and successor elements, either the element to be removed), then exchanged between peers. An operation always affects exactly one element: a block can be added to, or removed from the content. Upon receiving a remote operation, it is added to a queue and will be applied when it is causally ready, meaning that the preceding and following W-characters are part of the W-string in case of an *insert* operation, or the target W-character is part of the W-string in case of a *delete* operation.

When a document must be displayed, the WOOT algorithm computes a linearization of the W-string, and only the visible block are returned. This linearization is computed in such a way that the order of reception of the operations does not influence the result, as long as all the operations have been received.

3 The UniWiki Architecture

The central idea of our paper is to use a DHT system as the storage mechanism, with updates handled by running WOOT algorithm directly in the DHT nodes. This integration is possible by changing the behavior of the **put** and **get** methods. Instead of storing the actual wiki document as plain text content, each DHT node stores the corresponding W-String. On display, the DHT reconstructs a simplified version of the document, which is further processed by a wiki server front-end. On update, the wiki front-end creates a list of changes made by the user, which are transformed into WOOT operations that are applied inside the DHT.

Since a continuous synchronization is needed between all sites in a total replicated system, it is practical to restrict their number, as it is done, for instance, in the current architecture of Wikipedia. However, as the DHT allow accesses from many clients at the same time, in our approach wiki front-ends can be placed in numerous sites, without the need for any synchronization between them. Front-ends can be added or removed at any moment, because the entire data is stored outside the front-end, in the DHT. Also, the specifics of DHT systems allow adding and removing peers from the DHT even at high rates, thus opening the storage system for third-party contributions.

3.1 Overview

Basically, our system consists of two parts, as depicted in Figure 1: the DHT network, responsible for data storage, and the wiki front-ends, responsible for handling client requests.

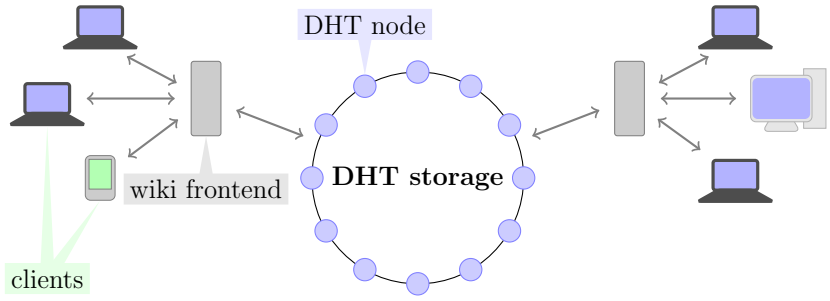

Figure 1: UniWiki architecture overview.

The **DHT storage network** is a hybrid network of dedicated servers and commodity systems donated by users, responsible for hosting all the data inside the wiki in a peer to peer manner. As in any DHT, each peer is assigned responsibility for a chunk of the key space to which resources are mapped. In our context, each peer is therefore responsible for a part of the wiki content. The information is stored on a peer in the form of a WOOT model. This allows to tolerate concurrent updates, occurring either when parallel editions arise from different access points, or when temporary partitions of the network are merged back. The WOOT algorithm ensures eventual consistency – convergence – on the replicated content stored by peers responsible for the same keys.

The **wiki front-ends** are responsible for handling client requests, by retrieving the WOOT pages for the requested page from the DHT, reconstructing the wiki content, rendering it into HTML, and finally returning the result to the client. In case of an update request, the front-end computes the textual differences corresponding to the changes performed by the user, transformes them into WOOT operations, then sends them to the DHT to be integrated, stored and propagated to the replicated peers.

3.2 Data model

As explained in section 2.2 and in [21], WOOT works not with the plain textual content of a document, but with an enriched version of it, in the form of a W-string. This is the **WOOT page model** that is stored inside the DHT, at each peer responsible for that document.

To update this model, **WOOT operations** computed from the user's changes are inserted in the DHT, first by calls to the **put** method, and then by inherent DHT synchronization algorithms.

Determining the actual changes performed by the user requires not just the new version of the content, but also the original version on which he started editing. And, since transforming textual changes into WOOT operations requires knowing the WOOT identifiers corresponding to each block of text, this means that the front-end must remember, for each user and for each edited document, the model that generated the content sent to the user. However, not all the information in the complete WOOT page model is needed, but just the visible part of the linearized W-string, and only the *id* and actual block content. This simplified W-string – or, from a different point of view, enriched content – is called the **WOOT page**, and is the information that is returned by the DHT and stored in the front-end.

When sending the page to the client, the front-end further strips all the meta-information, since the client needs only the visible text. This is the plain **wiki content** which can be transformed into HTML markup, or sent as editable content back to the client.

3.3 Algorithms

This section describes in term of algorithms the behaviors of a front-end server and of a DHT peer.

3.3.1 Behavior of a Front-end Server

The behavior of a front server can be summarized as follows. When a client wants to look at a specific wiki page, the method **onDisplay()** is triggered on the front-end server. First, the server retrieves the corresponding WOOT page from the DHT, using the hashed URI of the requested wiki content as the key. The received WOOT page is transformed into plain wiki content, which is rendered as HTML if necessary, then sent to the client.

```
onDisplay(contentURI)
  wootPage = dht.get(getHash(contentURI))
  wikiContent = extractContent(wootPage)
  htmlContent = renderHTML(wikiContent)
  return htmlContent
```

When a client requests a particular page in order to edit it, the method **onEdit()** is called on the front-end server. The WOOT page retrieved from the DHT is stored in the current editing session, so that the changes performed by the user can be properly detected (the *intention* of the user is reflected on the initial content displayed to him, and not on the most recent version, which might have changed). Then the wiki content is sent to the client for editing.

```
onEdit(contentURI)
  wootPage = dht.get(getHash(contentURI))
  session.store(contentURI, wootPage)
  wikiContent = extractContent(wootPage)
  return wikiContent
```

When a client terminates his editing session and decides to save his modifications, the method **onSave()** is executed on the front-end server. First, the old version of the wiki content is extracted from the current editing session. A classical textual differences algorithm [20] is used to compute modifications between this old version and the new version submitted by the client. These modifications are then

mapped[2] on the old version of the WOOT page in order to generate WOOT operations. Finally, these WOOT operations are submitted to the DHT.

```
onSave(contentURI, newWikiContent)
  oldWootPage = session.get(contentURI)
  oldWikiContent = extractContent(oldWootPage)
  changes[] = diff(oldWikiContent, newWikiContent)
  wootOps[] = ops2WootOps(changes[], oldWootPage)
  dht.put(getHash(contentURI), wootOps[])
```

3.3.2 Behavior of a DHT Peer

In order to comply to our architecture, the basic methods generally provided by a DTH peer have to be updated. Their behaviors differ from the basic behaviors provided by any DHT since the type of the value returned by a get request – a WOOT page – is not the same as the type of the value – a list of WOOT operations – stored by a put request.

When a `get` request is received by a DHT peer (which means that it is responsible for the wiki content identified by the targeted key), the method **onGet()** is executed. The WOOT page model corresponding to the key is retrieved from the local storage, the simplified WOOT page is extracted, and then sent back to the requester – generally, a front-end server.

```
onGet(key)
  wootPageModel = wootStore.get(key)
  wootPage = extractVisiblePage(wootPageModel)
  return wootPage
```

When a DHT peer has to answer to a `put` request, the method **onPut()** is triggered. First, the targeted WOOT page model is retrieved from the local storage. Then, each operation received within the request is integrated in the actual WOOT page and logged in the history of that page.

```
onPut(key, wootOps[])
  wootPageModel = wootStore.get(key)
  for (op in wootOps[])
      integrate(op, wootPageModel)
      wootPageModel.log(op)
```

Generally, DHTs provide a mechanism for recovering from abnormal situations such as transient or permanent failure of a peer, message drop on network, or simply new nodes joining. In such situations, after the execution of the standard mechanism that re-attribute the keys responsibility to peers, the method **onRecover()** is called for each key the peer is responsible for.

The goal of the **onRecover()** method is to reconciliate the history of a specific wiki content with the current histories of that content stored at other peers responsible for the same key. By the usage of the WOOT integration algorithm, reconciliating histories will also ensure convergence on the replicated WOOT models.

The method starts by retrieving the targeted WOOT page model and its history. Then, a digest of all the operations contained in this history is computed. Further, the method **antiEntropy()** is called on another replica – another peer responsible for the same key – in order to retrieve operations that the peer could have missed. Finally, each missing operation is integrated in the WOOT model and is added to its history.

```
onRecover(key)
  wootPageModel = wootStore.get(key)
  wootOps[] = wootPageModel.getLog()
  digests [] = digest(wootOps[])
  missingOps[] = getReplica(key).antiEntropy(digests[])
  for (op in missingOps[])
      integrate(op, wootPageModel)
      wootPageModel.log(op)
```

[2] A thorough description of the mapping algorithm is provided in [33].

```
onAntiEntropy(key, digests[])
  wootPageModel = wootStore.get(key)
  wootOps[] = wootPageModel.getLog()
  mydigests[] = digest(wootOps[])
  return notin(wootOps[], mydigests[], digests [])
```

4 Implementation

Over the last few years, many DHT implementations have been released. These systems take advantage of the computing at the edge paradigm, where resources available from any computer in the network can be used and are normally made available. However, contrasting with the client-server model, where few powerful servers provide all the computing and storage power to a much larger network of clients.

Nevertheless, such decentralized architectures also introduce new issues which have to be taken care of. Some of these issues include how to deal with constant node joins and leaves, network heterogeneity, and many others. Also, another important issue is the development complexity of new applications on top of this kind of networks. For these reasons, we need a middleware platform that provides the necessary abstractions and mechanisms to construct distributed applications.

Following the ideas developed in previous sections, we want to implement our algorithms over a DHT system. In this line, we summarize the objectives of our implementation in three points:

- Modify the put and get DHT methods behavior.
- Establish replication strategies and handle replicas.
- Add consistency logic in these DHT mechanisms.

In this setting, another important aspect is that we want to change the behavior of the DHT system. To address this problem, there are different options, but it is clearly a good scenario to apply distributed interception mechanisms.

The benefits of this approach will be:

- Full control of the DHT mechanisms, including runtime adaptations.
- Decoupled architecture between wiki front-end and DHT sides.
- Legacy and transparency for wiki front-end applications.

Lastly, we propose the use of a distributed P2P interception middleware, called Damon [17], that fulfills the DHT, middleware, and interception requirements.

4.1 Damon Overview

By making efficient use of DHT substrates and dynamic interception mechanisms, Damon is designed as a fully distributed interception middleware. Its main goal is to provide the necessary abstractions and services to develop distributed P2P interceptors. A description of them is shown as follows. However, for more detailed information on Damon, please see [16, 17]

Mainly, Damon provides new features (i.e. scalability) to existent or new applications transparently. Therefore, distributed interception presents a different philosophy than traditional distributed solutions like remote object or component models. Usually, when one designs a new application, he creates the main aspect without any distributed concern in mind. Simply, we may design the main concern first by *thinking in local*, and later implementing the rest of the distributed interception application designing the necessary concerns (i.e. distribution).

The main features of Damon are:

- **P2P Routing**: taking advantage of the built-in DHT-based P2P network, Damon allows key-based, direct-node and neighbor routing abstractions.
- **Distributed Composition Model**: this model allows connection-oriented features. In addition, the composition model allow us to make a good abstraction, clear interaction scheme, and reconfiguration in runtime.

- **Distributed Interception**: this mechanism was introduced to allow source hooks for local interception, and remote calls for distributed purposes. Another important feature, is which Damon supports interception of their own remote connectors. This own interception can be performed before, during, and after the call routing.

Following these ideas, developers can implement and compose distributed interceptors in large-scale environments. When the local interception (source hook) is peformed, these distributed interceptors trigger the remote calls via P2P routing abstractions.

4.2 UniWiki Implementation

Like traditional wiki application (i.e. Wikipedia), we have the local execution of our wiki front-end. This scenario is ideal to apply distributed interception, because we can intercept the local behavior to extend/compose the necessary concerns. Thereby, using Damon we are able to inject our algorithms of distribution, replication, and consistency transparently.

In Figure 2 we can see the UniWiki source hook, where we aim to intercept locally the typical wiki methods of store and retrieve (in this case we use a generic example), in order to distribute remotely. In addition, the source hook solution helps to separate local interception, interceptor code, and the wiki interface. On the other hand, source hooks have other benefits like a major level of abstraction, or degree of accessibility for distributed interceptors.

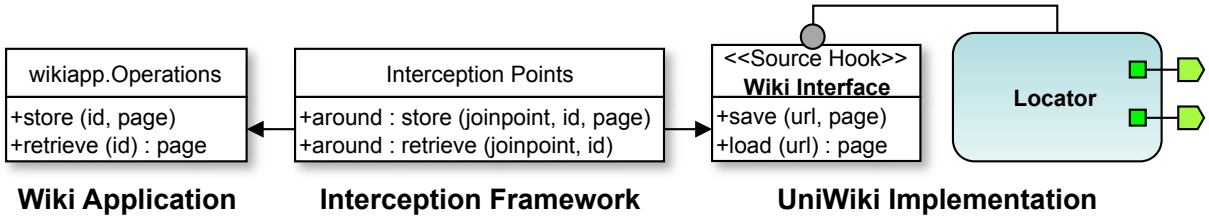

Figure 2: Wiki Source Hook Interface.

Following such approach, integration with other wiki applications is quite simple, and can be easily used for third party wiki applications, including different versions, transparently.

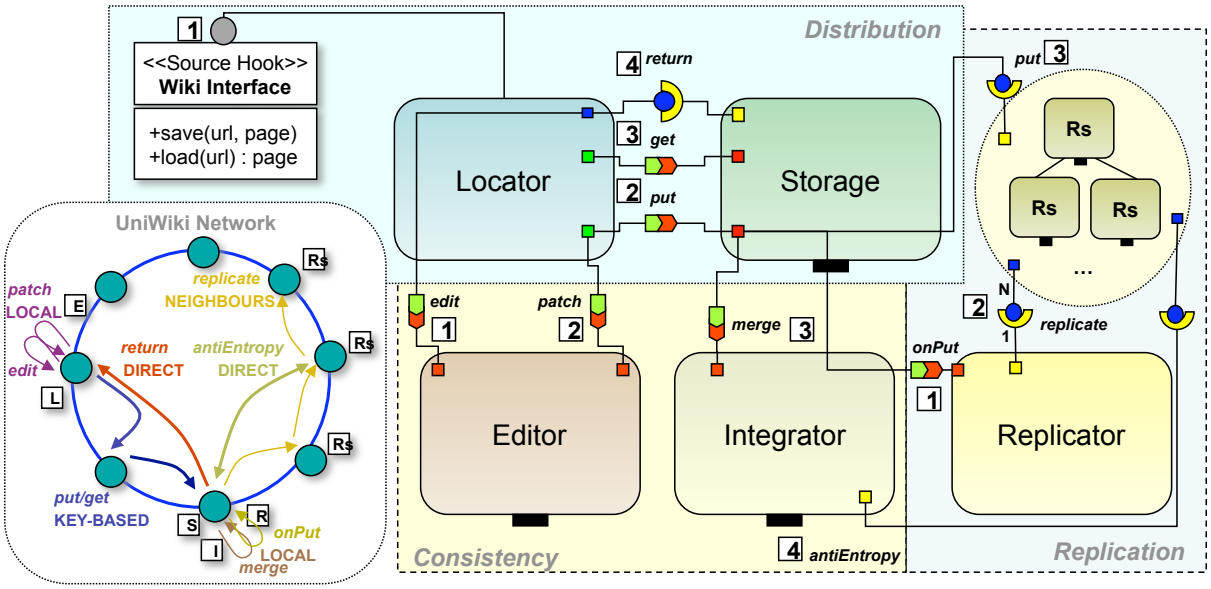

Figure 3: UniWiki composition diagram.

We now describe the UniWiki execution step by step as shown in Figure 3, focusing on the integration of the algorithms and the interaction of the different concerns. In this line, we analyze the context, and extract three main concerns that we need to implement: distribution, replication and consistency.

Obviously, distribution is the main concern, and in our solution uses the key-based routing abstraction. The replication concern is also based on P2P mechanisms, and follows the neighbors strategy. Finally, as we see in previous section, the consistency concern remains on the WOOT algorithm. In this implementation, it allows edition, patching, and merging of wiki pages, and it performs these operations via distribution calls interception.

Distribution:

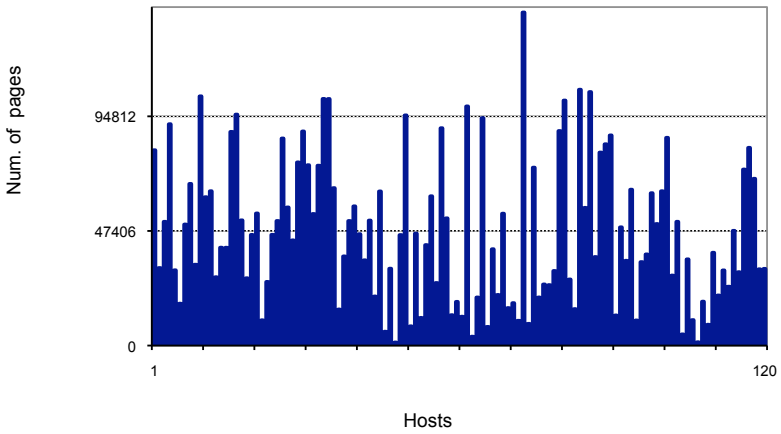

Figure 4: Empirical results - Distribution.

1. The starting point of this application is the wiki interface used by the existing wiki application. We therefore introduce the **Wiki Interface** source hook that intercepts the save, and load methods. Afterward, the Locator distributed interceptor is deployed and activated on all nodes of the UniWiki network. Its main objective is to locate the responsible node of the local insertions and requests.

2. These save executions are propagated using the **put** connector. Consequently, the remote calls are routed to the key owner node, by using their url to generate the key through the key-base routing.

3. Once the key has reached its destination, the registered connectors are triggered on the Storage instance running on the owner host. This distributed interceptor has already been activated on start-up on all nodes. For request case (**get**), the idea is basically the same, with the Storage receiving the remote calls.

4. Furthermore, it propagates later an asynchronous response using the **return** call via direct node routing. Finally, the get values are returned to the Locator originator instance, using their own connector.

Once we have the wiki page distribution running, we may add new functionalities as well. In this sense, we introduce new distributed interceptors in order to extend or modify the current application behavior in runtime. More specifically, these instances add new important services (replication and consistency) as shown right away.

Replication:

1. When dealing with the save method case, we need to avoid any data storage problems which may be present in such dynamic environments as large-scale networks. Thus, data is not only to be stored on the owner node, because if this host leaves the network for any reason, its data would surely become unavailable. In order to address this problem, we activate the Replicator interceptor in runtime, which following a specific policy. The Replicator has a connector called **onPut**, which intercepts the Storage put requests from the Locator service in a transparent way.

2. Thus, when a wiki page insertion arrives to the Storage instance, this information is re-sent (**replicate**) to the ReplicaStore instances activated in the closest neighbors.

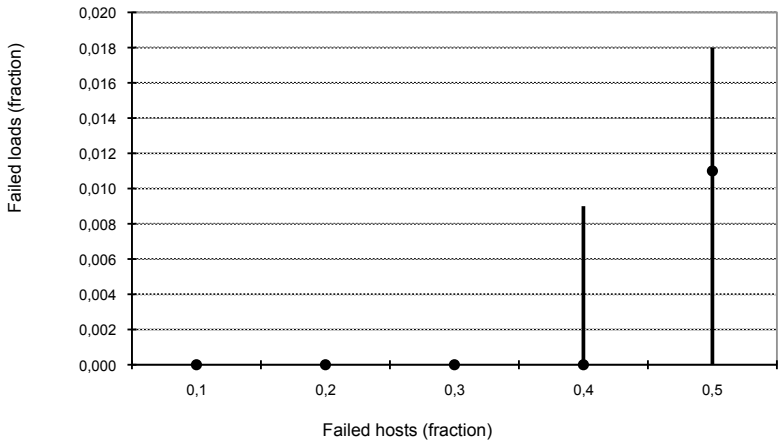

Figure 5: Empirical results - Replication.

3. Last but not least, ReplicaStore distributed interceptors are continuously observing the state of the copies that they are keeping. If one of them detects that the original copy is not reachable, it re-inserts the object again, using a remote connector **put**, in order to replace the Locator remote call.

Consistency:

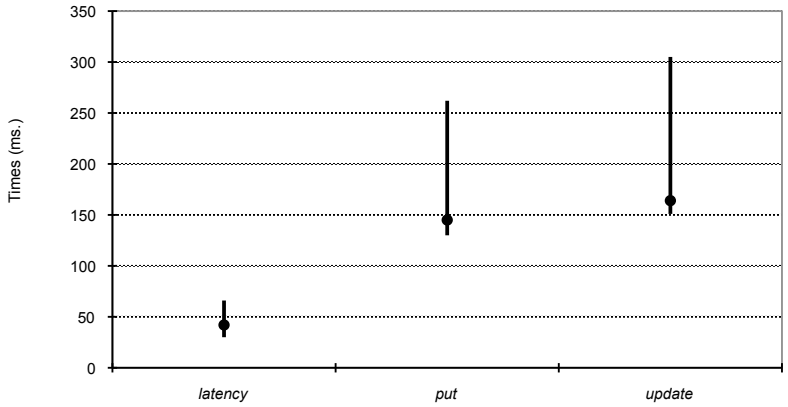

Figure 6: Empirical results - Consistency.

Based on the WOOT framework, we create the Editor (situated on the front-end side) and the Integrator (situated on the back-end side) distributed interceptor, those intercept the DHT-based calls to perform the consistency behavior. Their task is the modification of the distribution behavior, adding the patch transformation in the edition phase, and the patch merging in the storage phase.

1. The Editor distributed interceptor owns a connector (**edit**) that intercepts the return remote calls from Storage to Locator instances. This mechanism stores the original value in a session. Obviously, in a parallel way, the Integrator prepares the received information to be rendered as a wiki page.

2. Later, if the page is modified, a save method triggers the put mechanism, where another connector (**patch**) transforms the wiki page into the patch information by using the saved session value.

3. In the back-side, the Integrator instance intercepts the put request, and **merge**s the new patch information with the back-end contained information. The process is similar to the original behavior, but changing the wiki page with a consistent patch information.

4. In this setting, having multiple replicated copies, leads to inconsistencies. We use the **antiEntropy** technique, in order to recover a log of differences among each page and their respective replicas. Finally, the Integrator sends the necessary patches to be sure that all copies remain consistent.

5 UniWiki Evaluation

We have conducted several experiments to measure the viability of our UniWiki system. We have used Grid'5000 [5]. The Grid'5000 platform is a large-scale distributed environment that can be easily controlled, reconfigured and monitored. The platform is built with 5000 CPUs distributed over 9 sites in France. Every site hosts a cluster and all sites are connected by a high speed network.

In this sense, we conducted the experiments using 120 real nodes from the Grid'5000 network, located in different geographical locations, including Nancy, Rennes, Orsay, Toulouse, and Lille.

Finally, for our prototype we chose the DHT Implementation of Pastry (FreePastry) for its efficient Java implementation. In addition, for local interception we use an Aspect-Oriented Programming (AOP) [3] framework.

5.1 Testbed

When analyzing our large-scale system we have three main concerns: load-balancing in data dispersion among all nodes (distribution), failed hosts that do not affect system behavior (replication), and the operation performance (consistency).

For distribution, we study the data dispersion in the network.

- *Objective*: demonstrate that when our system works with a high number of hosts, is able to store a real large data set of wiki pages, and that all the information is uniformly distributed among them.
- *Description*: create a network of 120 hosts, and using a recent Wikipedia snapshot, we introduce their 5,688,666 entries. The idea is that data distribution is uniform, and each host has a similar quantity of values (wiki pages).
- *Results*: we can see in Figures 4 that we have a system working and the results are as expected. Thereby, the distribution of data trends to be uniform. Results indicate that each host has an average of 47,406 stored wiki pages, and using an approximate average of space (4.24 KB) per wiki pages we have 196 MB, with a maximum of 570.5 MB (137,776 values) in one node.
- *Why*: Uniform distribution of data is guaranteed by the key-based routing substrate, and by the hash function (SHA-1) employed to generate the keys. However, with this number of hosts (120) the distribution values show that a 54.17% are over the average, and a 37.5% are over the double of the average. Furthemore, we can see similar results in simulations (using PlanetSim [22]), with a random distribution of 120 nodes.

  We make another simulation with 1000 nodes using the same DHT simulator. The results of this last simulation (Figure 7) shows that the values are: 63.7% over average and 20.6% over double of the average, because with a high number of nodes the uniformity is improved.

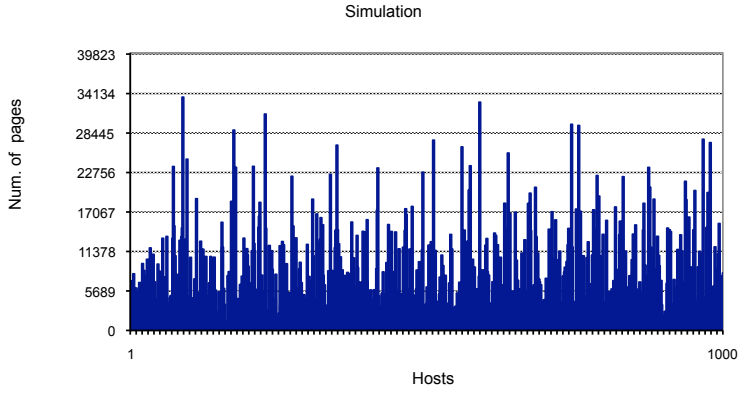

Figure 7: Wikipedia data distribution.

Secondly, we study the fault-tolerance of our system platform.

- *Objective*: demonstrate that in a real network with failures, our system continues working as expected.

- *Description*: In this experiment we introduce problems on a specific fraction of the network. In this case, each host inserts 1000 wiki pages, and retrieves 1000 on them randomly. The objective of this test is to check persistence and reliable properties of our system. After the insertions, a fraction of the hosts fail without warning, and we try to restore these wiki pages from a random existing host.
- *Results*: We can see the excellent results in Figure 5. Even in the worst case (50% of network fails at the same time), we have a high probability (average of 99%) to activate the previously inserted wiki pages.
- *Why*: The theoretical probability that all the replicas for a given document fail is $\prod_{i=1}^{n} r_i = r_1 * r_2 * ... * r_n = (r)^n$ where n is the replication factor and r the probability that a replica fails. For instance, when r = 0.5, and n = 6, the result is $(0.5)^6 \approx 0.015 \approx 1.5\%$.

Finally, for consistency we study the performance of our operations.

- *Objective*: demonstrate that our routing abstraction is efficient in terms of time and network hops.
- *Description*: similar to the previous experiment, we create a 120 hosts network, and each host inserts and retrieves 1000 times the same wiki page. The idea is that the initial value is modified, and retrieved concurrently. In this point, we make this experiment twice. The first time with consistency mechanisms disabled (only the put call), and the second time with these mechanisms enabled (patch and merge). In the last step, the consistency of the wiki page is evaluated for each host.
- *Results*: for this experiment, we found that the consistency is guaranteed by our system. We can see the operation times in Figure 5, and for each operation the number of hops has an average of 2. Therefore, the wiki page put operation has an average time of **145** ms, and an overall overhead of: **3.45**. For update operations the value is logically higher : **164** ms. with an overhead of **3.90**. Finally, the update operation overhead respects put operation, when the consistency operation is performed, is **1.13**.
- *Why*: Due to the nature of the Grid experimentation platform, latencies are low. Moreover, we consider that the operation overhead is also low. Finally, theoretical number of hops is logarithmic respect the size of the network. In this case, log (120 hosts) = 2 hops.

6 Conclusions and Future Work

In this paper we propose an efficient P2P system for storing distributed wikis, extended to large-scale scenarios transparently. In this line we present the algorithms, a prototype, and the experimentation.

Nowadays, many solutions for wiki distribution are proposed, but as we can see in the Section 2.1, the current approaches focus on only one half of the problems, failing to address both efficient distribution and correct replication. We combine two intensively studied technologies, each one addressing a specific aspect: distributed hash tables are used as the underlying storage and distribution mechanism, and WOOT ensures that concurrent changes are correctly propagated and merged for every replica.

We propose a completely decoupled architecture, where our solution is totally abstracted from the real wiki application. In our approach we define the storage behavior from the scratch, apply this behavior on an existing DHT library, and the wiki presentation and business logic is full provided by the wiki application. Therefore, for our implementation we use a distributed interception middleware over DHT-based networks, called Damon.

Validation of UniWiki has been conducted on the Grid'5000 testbed. We have proved that our solution is viable in large-scale scenarios, and that the system has acceptable performance. Our experiments where conducted with real data [35] from Wikipedia which include almost 6 million entries.

Finally, UniWiki can be downloaded at `http://uniwiki.sourceforge.net/`, under a LGPL license.

In the near future, we plan to apply our approach to a more sophisticated front-end, the XWiki application server. Other future directions include studying how to manage security access on wiki applications deployed on our proposed peer-to-peer network, and how to perform search.

7 Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, an initiative from the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS and RENATER and other contributing partners (see https://www.grid5000.fr/).

This work has been partially funded by the Spanish Ministry of Science and Technology through project P2PGRID TIN2007-68050-C03-03.

References

[1] L. Allen, G. Fernandez, K. Kane, D. Leblang, D. Minard, and J. Posner. ClearCase MultiSite: Supporting Geographically-Distributed Software Development. In J. Estublier, editor, *Software Configuration Management: Selected Papers of the ICSE SCM-4 and SCM-5 Workshops*, number 1005 in Lecture Notes in Computer Science, pages 194–214. Springer-Verlag, Oct. 1995.

[2] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, Dec. 2004.

[3] Aspect-Oriented Software Development Community. http://aosd.net/.

[4] M. Bergsma. Wikimedia architecture. http://www.nedworks.org/~mark/presentations/san/Wikimedia%20architecture.pdf, 2007.

[5] R. Bolze, F. Cappello, E. Caron, M. Dayd, F. Desprez, E. Jeannot, Y. Jgou, S. Lantri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche. Grid'5000: A Large Scale and Highly Reconfigurable Experimental Grid Testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, Nov. 2006.

[6] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. *Lectures Notes in Computer Science*, 2735:33–44, 2003.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - SOSP 2007*, pages 205–220. ACM Press, 2007.

[8] B. Du and E. A. Brewer. Dtwiki: a disconnection and intermittency tolerant wiki. In *Proceeding of the 17th international conference on World Wide Web - WWW 2008*, pages 945–952, New York, NY, USA, 2008. ACM Press.

[9] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[10] K. Fall. A Delay-tolerant Network Architecture for Challenged Internets. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM 2003*, pages 27–34. ACM Press, 2003.

[11] Git – fast version control system. http://git.or.cz/.

[12] L. Gong. JXTA: A Network Programming Environment. *IEEE Internet Computing*, 5(3):88–95, 2001.

[13] B. Kang. Summary Hash History for Optimistic Replication of Wikipedia. http://isr.uncc.edu/shh/.

[14] S. Ktari, M. Zoubert, A. Hecker, and H. Labiod. Performance Evaluation of Replication Strategies in DHTs under Churn. In *Proceedings of the 6th International Conference on Mobile and Ubiquitous Multimedia - MUM 2007*, pages 90–97, Oulu, Finland, Dec. 2007. ACM Press.

[15] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System based on the XOR Metric. *Lecture Notes In Computer Science*, 2429:53–65, 2002.

[16] R. Mondéjar, P. García, C. Pairot, and A. Skarmeta. Damon: a decentralized aspect middleware built on top of a peer-to-peer overlay network. In *Proceedings of the 6th international workshop on Software engineering and middleware*, pages 23–30, Portland, Oregon, USA, 2006. ACM Press.

[17] R. Mondéjar, P. García, C. Pairot, and A. Skarmeta. Building a distributed AOP middleware for large scale systems. In *Proceedings of the 2008 workshop on Next generation aspect oriented middleware*, pages 17–22, Brussels, Belgium, 2008. ACM Press.

[18] J. C. Morris. Distriwiki: A distributed peer-to-peer wiki network. In *Proceedings of the 2007 International Symposium on Wikis - WikiSym 2007*, pages 69–74, New York, NY, USA, 2007. ACM Press.

[19] P. Mukherjee, C. Leng, and A. Schürr. Piki - a peer-to-peer based wiki engine. *Proceedings of the IEEE International Conference on Peer-to-Peer Computing - P2P 2008*, 0:185–186, 2008.

[20] E. W. Myers. An O(ND) Difference Algorithm and its Variations. *Algorithmica*, 1(2):251–266, 1986.

[21] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 259–267, Banff, Alberta, Canada, Nov. 2006. ACM Press.

[22] PlanetSim. http://planetsim.sourceforge.net/.

[23] S. Plantikow, A. Reinefeld, and F. Schintke. Transactions for Distributed Wikis on Structured Overlays. In *Managing Virtualization of Networks and Services*, volume 4785, pages 256–267. Springer-Verlag, 2007.

[24] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The Bittorrent P2P File-Sharing System: Measurements and Analysis. *Lecture Notes in Computer Science*, 3640:205, 2005.

[25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A Scalable Content-addressable Network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM'01*, pages 161–172. ACM Press, 2001.

[26] RepliWiki – A Next Generation Architecture for Wikipedia. http://isr.uncc.edu/repliwiki/.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Lecture Notes In Computer Science*, 2218:329–350, Nov. 2001.

[28] Y. Saito and M. Shapiro. Optimistic Replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[29] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[30] C. Sun and C. A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW'98*, pages 59–68, New York, NY, USA, Nov. 1998. ACM Press.

[31] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, Mar. 1998.

[32] G. Urdaneta, G. Pierre, and M. van Steen. A decentralized wiki engine for collaborative wikipedia hosting. In *Proceedings of the 3rd International Conference on Web Information Systems and Technologies*, 2007.

[33] S. Weiss, P. Urso, and P. Molli. Wooki: a P2P Wiki-based Collaborative Writing Tool. *Lecture Notes In Computer Science*, 4831(1005):503–512, Dec. 2007.

[34] Wikipedia, the online encyclopedia. http://www.wikipedia.org/.

[35] Wikipedia Data, 2008. http://download.wikimedia.org/enwiki/latest/.

[36] Wikipedia Statistics, 2008. http://meta.wikimedia.org/wiki/Statistics.

[37] Code Co-op. http://www.relisoft.com/co_op/.

[38] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2001.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399